CHIP PLUS

EINE BEILAGE
IN CHIP NR. 8
AUGUST 1987

PASCAL MODULA

Niklaus Wirth, der Schöpfer von Modula-2: „Gute Sprachen kommen aus einem einzigen Kopf"

Gregor Henger

Die Geschichte von Modula-2 läßt sich nicht darstellen, ohne auf ihre Vorgängersprache einzugehen, auf Pascal. Und wer von Modula-2 oder von Pascal spricht, muß sich mit dem Schöpfer dieser Sprachen beschäftigen, mit dem Schweizer Informatiker Niklaus Wirth.

Der Anstoß zur Entwicklung von Modula-2 ist einem Forschungsprojekt am Institut für Informatik der Eidgenössischen Technischen Hochschule zu verdanken. Die Projektidee war Wirth in Palo Alto gekommen, wo er am Research Center von Rank-Xerox einen Forschungsurlaub verbracht hatte. Wirth und seine Mitarbeiter am Institut für Informatik nahmen sich 1977 vor, ein integrales System, also Hardware und Software eines neuen Rechners, zu entwickeln.

Als technologischer Leckerbissen sollte das neue System in einer einzigen Sprache zu programmieren sein – von den Mikroprogrammen des Prozessors bis hinauf zu den Applikationen. „Wenn eine Sprache sich als weitgehend ungeeignet für eine Anwendung herausstellt, die von ihrem Autor nicht vorausgesehen war", so sagte Wirth, „sollten wir den Mut haben, ein neues, wirklich geeignetes Werkzeug zu bauen, anstatt einfach ein neues Pflaster auf das alte zu kleben." Und er praktizierte, was er predigte: Pascal war für die Programmierung der neuen Maschine wenig geeignet, und so schuf Wirth Modula-2. Diesen Namen erhielt die Sprache, weil sie in ihrer Syntax mehr einer Modula genannten Sprache glich, die Wirth bei Multiprogramming-Experimenten zwei Jahre zuvor entwickelt hatte, als dem viel älteren Pascal.

Der erste Modula-2-Compiler wurde für den PDP-11-Prozessor von Digital Equipment Corporation (DEC) geschrieben. Seither sind eine ganze Reihe weiterer Compiler entstanden, von denen viele ihre geistige Verwandschaft mit dem Modula-2-„Urcompiler" als Qualitätsausweis betonen. 1986 hat Wirth jedoch einen Markstein gesetzt, an dem sich neue Modula-2-Implementationen messen lassen müssen: seinen Single-Pass-Compiler für Modula-2.

Ein Programm, das in einer höheren Programmiersprache – also etwa Pascal oder Modula-2 – geschrieben wurde, bildet zunächst ein Textfile, einen Datenstrom, der aus ASCII-Zeichen (ASCII: American Standard Code for Information Interchange) besteht. Um in einem Prozessor ausführbar zu werden, muß es in binäre Form, sogenannten Objekt-Code, übersetzt werden. Dies besorgt der Compiler. Diese Compiler gehören zu den technisch anspruchsvollsten Programmen, die es gibt. In der Regel erfolgt der Übersetzungsvorgang nicht auf einmal, sondern in mehreren Durchgängen durch das Quellencode genannte Textfile. Wirth ist es gelungen, einen Compiler zu schreiben, der den Übersetzungsvorgang in einem Durchgang bewältigt, einem darum so genannten Single-Pass-Compiler. Ein Single-Pass-Compiler arbeitet um ein Vielfaches schneller als ein Mehr-Pass-Compiler, der mehrere Durchgänge durch den Quellencode machen muß, und ist obendrein zuverlässiger.

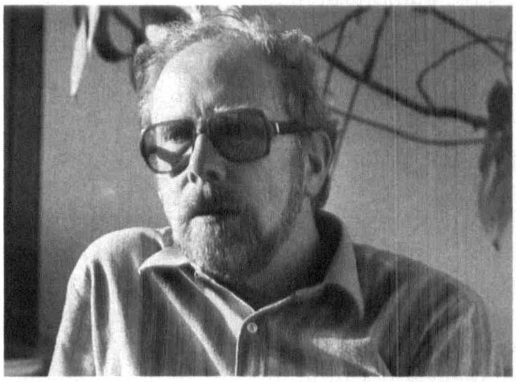
Niklaus Wirth erwarb sein Rüstzeug in den USA

Der Mann, der dieses technische Kabinettstück zuwege brachte, wurde 1934 in der Industriestadt Winterthur unweit von Zürich geboren und wuchs im Schatten der Kamine der Dieselmotorenfabrik Sulzer und der schweizerischen Lokomotivfabrik auf. Zu seinen ersten Kontakten mit höherer Technik gehörte die Beschäftigung mit funkgesteuerten Modellflugzeugen – der junge Niklaus Wirth muß zu den ersten Besitzern der damals noch teuren Anlagen in der Schweiz gehört haben. So war es kein Wunder, daß sich der jugendliche Radiofan nach dem Abitur bei der Eidgenössischen Technischen Hochschule (ETH) im nahen Zürich zum Studium der Elektrotechnik einschrieb.

Das war Mitte der fünfziger Jahre und in jener Zeit gehörte das Polytechnikum am Zürichberg zu den ersten Institutionen der Welt, die elektronische Rechner bauten und betrieben, die den Namen Computer verdienten. Der Kreis um die Professoren

Rudolf Stiefel, Walter Spyser und Hans Rutishauser stand in jenen Tagen an der vordersten Front eines Fachgebietes, das erst viel später den Namen Informatik erhalten sollte. Aber ihre Pionierarbeiten waren Forschungstätigkeit. Niklaus Wirth kam mit diesen Anlagen nicht in direkten Kontakt. Obwohl die ETH Zürich neben dem Massachussetts Institute of Technology (MIT) und der Universität California in Berkeley eine

Niklaus Wirth



der drei führenden Informatik-Institutionen der Welt war, konnte Niklaus Wirth nach seinem Ingenieurdiplom nicht in Zürich Computerwissenschaft studieren – es fanden keine entsprechenden Lehrveranstaltungen statt.

So zog der Elektroingenieur 1959 nach Kanada, an die Université Laval in Quebec. Dort erwarb er seinen Master of Science. 1960 wechselte er nach Berkeley, an die renommierte University of California. Diese Institution gehörte zu den ersten Universitäten, an denen Informatik gelehrt wurde und Wirth erwarb dort seinen Doktorgrad. Schon damals konzentrierte sich sein Interesse auf Computersprachen. Neben dem seit 1956 bekannten FORTRAN waren 1960 die ersten ALGOL- und COBOL-Versionen vorgestellt worden. Sie gehen übrigens auf wesentliche, an der ETH Zürich entstandene Entwicklungen zurück und erlaubten erstmals die Technik des strukturierten Programmierens.

1963 nahm Wirth eine Stelle als Assistant Professor an der Stanford University an und zog ans südliche Ende der Oakland Bay, ans untere Ende des heute als Silicon Valley bekannten Tales. Dort entstand Wirths erste eigenständige Sprache, PL360, die auf den IBM-360-Computer zugeschnitten war.

Frust im Komitee

In Fachkreisen war der Schweizer bekannt und die International Federation for Information Processing (IFIP) berief ihn in eine Gruppe von Sprachexperten, die eine Nachfolgeversion von ALGOL-60 erarbeiten sollte. Die Arbeit in diesem Komitee war für Wirth zwar nicht immer erfreulich, doch aus ihr ging ALGOL-W hervor. Im IFIP-Komitee traf Wirth den englischen Wissenschaftler Charles Hoare und die beiden arbeiteten zusammen, um eine revidierte und angereicherte ALGOL-Version zu entwickeln. Doch das Komitee konnte sich nicht zur Annahme des Vorschlages durchringen. Sie arbeiteten zwar weiter in dem Gremium mit, entwickelten jedoch ihren Vorschlag zur 1966 publizierten Version ALGOL-W weiter.

Wirth erinnert sich: ,,PL360 war eigentlich ein Nebenprodukt meiner Mitarbeit im IFIP-Komitee. Für die Implementation des ALGOL-W-Compilers hatten wir die Wahl zwischen Assemblersprache und FORTRAN – keine dieser Alternativen war besonders verlockend. Darum schrieb ich PL360 und implementierte diese Sprache als Tool für ALGOL-W." Das war 1965. ALGOL-W spiegelt Wirths Streben nach Ordnung und Systematik wider. Die Sprache veranlaßt den Programmierer, sein Problem explizit darzulegen und auf nicht deklarierte Voraussetzungen zu verzichten. Die Leistungsfähigkeit von ALGOL-W ist in der umfassenden Einfachheit der Sprache begründet.

Das IFIP-Komitee tagte noch drei Jahre lang weiter. In jeder Sitzung wurde der Entwurf länger – kaum ein Komiteemitglied wollte darauf verzichten, persönliche Steckenpferde zu reiten und daraus resultierende Zusätze einzubringen. Mit jeder Sitzung platzte wieder eine Illusion Wirths – die Sprache wurde zu einem Krebsgeschwür der Komplexität. 1968 endlich ,,wurde das Monster geboren", wie ein Komitee-Mitglied sagte und erhielt den Namen ALGOL-68.

Geburt und Siegeszug von Pascal

Die Diskussionen um ALGOL-68 verstärkten eine Einsicht, die Wirth schon zuvor gewonnen hatte: ,,Gute Sprachen kommen aus einem einzigen Kopf." Die ALGOL-68-Arbeiten waren noch nicht abgeschlossen, als Wirth 1967 einem Ruf aus der alten Heimat folgte, an die Universität Zürich, wo er eine Assistenzprofessur in der Fachgruppe für Computerwissenschaften übernahm. Ein Jahr später wurde ihm eine Doppelprofessur angeboten – an der Universität Zürich und an der ETH. Wirth – der in seiner Antrittsvorlesung übrigens die Probleme vorhersagte, mit welchen heute die Schweiz bei der verbreiteten Anwendung

der Informatik kämpft – erfüllte sich einen langgehegten Wunsch. „Ich hatte das Gefühl, daß ich eine strukturierte Sprache brauche, um das zu tun, was mich wirklich interessiert – Compiler und Systemprogramme schreiben." So entstand eine Sprache, die klare didaktische Konzepte mit rigoroser Struktur verband: Pascal, nach jenem französischen Philosophen benannt, der schon als Junge seine Kopfschmerzen mit dem Lösen geometrischer Probleme bekämpfte und 1639 eine der ersten Rechenmaschinen baute.

Pascal enthält eine Reihe damals neuartiger Konzepte, vor allem diejenigen des Datentyps und des Recordhandling. Damit wurde es zum ersten Male möglich, Datenstrukturen in gleicher Weise zu konstruieren wie Algorithmen. Wirth implementierte den ersten Pascal-Compiler auf der großen CDC 6600-Maschine der Zürcher ETH im Jahre 1970. Er sah in seiner neuen Sprache in erster Linie ein Unterrichts-Hilfsmittel und hatte sie so aufgebaut, daß sie praktisch vollständig maschinenunabhängig war. Was in den kommenden zehn Jahren mit Pascal geschah, grenzt in unserer kommerziell orientierten Welt an ein Wunder. Pascal hatte keinerlei Unterstützung durch irgendwelche Herstellerfirmen oder Fachorganisationen. Doch innerhalb von zehn Jahren wurde Pascal zur beliebtesten Sprache der Welt für professionelle Programmierer auf mittleren und kleinen Rechnern.

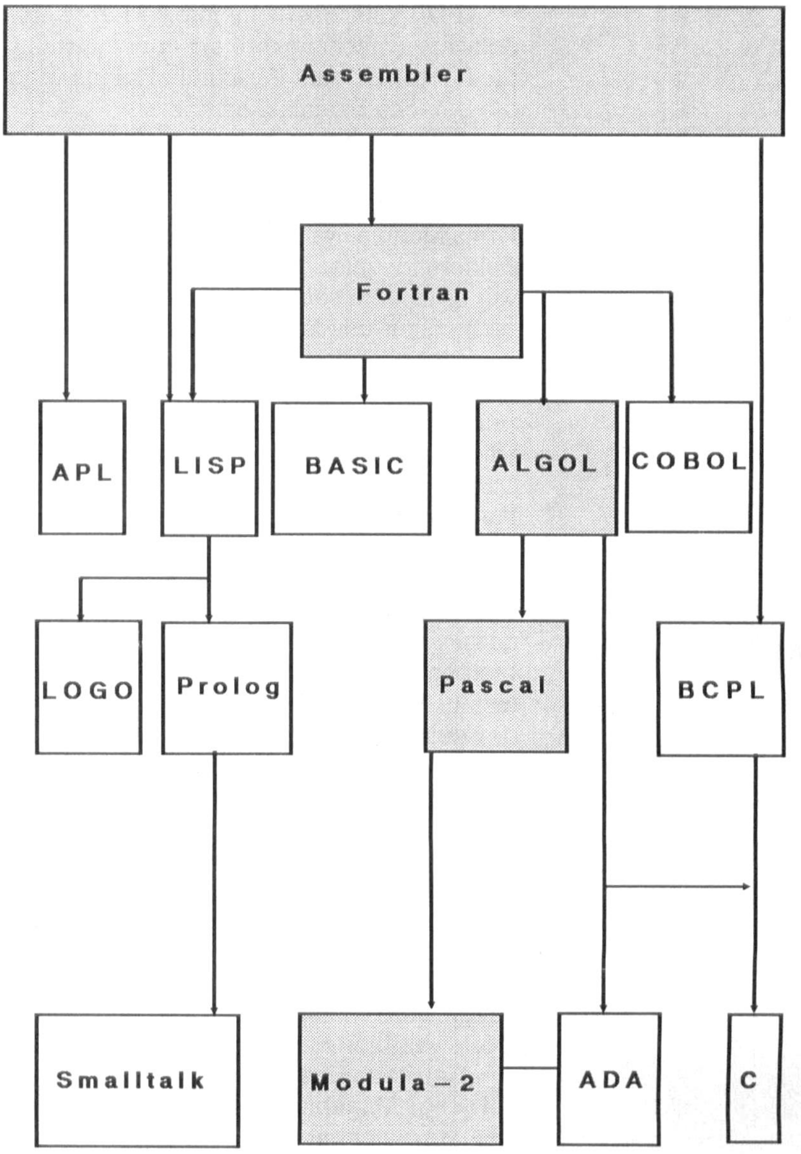
Wesentlichen Anteil an diesem Erfolg der neuen Sprache hatte auch eine Entwicklung der University of California at San Diego, das UCSD-System mit seiner P-Maschine, womit die Sprache auf einfache Weise auf Mikrocomputern – unter anderem auf dem damaligen Marktführer Apple II – zu implementieren war.

Im Jahre 1975 setzte Niklaus Wirth einen weiteren Markstein. Er veröffentlichte sein Lehrbuch „Algorithmen und Datenstrukturen". Heute in die Sprachen der meisten Industriestaaten übersetzt, ist dieses Werk zur Bibel für eine ganze Generation von Informatikern geworden. Wirths Prosa kann kaum flüssig genannt werden, dafür aber entschädigt er seine Leser mit zahllosen Programmbeispielen in Pascal von kristallener Klarheit, großer Eleganz und häufig viel Witz.

Pascal, als rechnerunabhängiges Unterrichts-Hilfsmittel konzipiert, hatte für professionelle Programmierer auch einige wesentliche Nachteile. Es war nicht für die Programmierung größerer Softwaresysteme gedacht, darum fehlte ein Modularisierungskonzept. Die Möglichkeiten zur Textverarbeitung waren nicht ausreichend, für maschinennahe Programmierung mußten in der jeweiligen Assemblersprache geschriebene Routinen eingebunden werden, es gab keine Sprachmittel zur Beschreibung der Nebenläufigkeit und Synchronisation von Prozessen.

Daß Pascal trotz dieser Einschränkungen Hörsäle und Laboratorien in aller Welt eroberte und sowohl in kleinsten wie größten Software-Entwicklungsfirmen eingesetzt wurde, „übertraf meine wildesten Träume", sagt Wirth, „ich hatte mir nie überlegt, wie erfolgreich Pascal kommerziell sein könnte. Ich hoffte lediglich, daß die Sprache akzeptiert würde, insbesondere die Ideen dahinter. Die Tatsache, daß Pascal selbst so weite Verbreitung fand – das war eine nette Überraschung."

Der Stammbaum der Programmiersprachen zeigt die Entwicklung von 1950 bis 1980



Das ultimative Kompliment für sein Pascal erhielt Niklaus Wirth von einer Stelle, mit der ihn nichts verbindet, die aber weltgrößter Computeranwender ist: das Department of Defense (DOD) der US-Regierung. Das DOD ließ – einmal mehr von einem Komitee – eine Sprache entwickeln, die ausdrücklich auf Pascal basieren sollte. Wirths Vorbehalte gegen von Komitees geschaffene Sprachen wurden bestätigt: ADA, die vom DOD 1979 sanktionierte Sprache, geriet zum Weißen Elefanten.

Wirth, der seit seinen ALGOL-68-Erfahrungen alle Einladungen in Sprachschöpfungs-Gremien abweist, hatte auch dem DOD einen Absagebrief geschickt. Mit sanfter Häme im Ton stellte Wirth nachträglich fest: „Der erste ADA-Compiler hatte 50 000 Zeilen Quellencode – unser Modula-2-Compiler nur 5000 Zeilen."

Ausflüge in die Hardware

1978 kam das kombinierte Hardware-Software-Forschungsprojekt des Instituts für Informatik zum Abschluß. Daraus ging die Sprache Modula-2 hervor.

Richard Ohran, heute selbst Professor an der Brigham Young University im Staate Utah und Vice President of Engineering der Firma Modula Systems, arbeitete damals als Doktorand unter Wirth und erinnert sich, wie es zu dem seltsamen Namen kam: „Wir zerbrachen uns lange Zeit die Köpfe über einen geeigneten Namen. Schließlich kam Wirth eines Tages und sagte, er habe beschlossen, die Maschine Lilith zu nennen. Lilith, so erklärte Wirth, sei ein weiblicher Dämon in der jüdischen Mythologie, eine Frau, so verführerisch schön wie Gott sie nur schaffen konnte – aber sie hatte keine Seele. Nach der Legende verfolgte Lilith des nachts die Männer und verleite sie dazu, von ihren Frauen und Kindern fernzubleiben." Im vergangenen Jahr wurde am Institut für Informatik ein Rechner gebaut, der eine Weiterentwicklung von Lilith darstellt – anstelle des ursprünglichen Bit-Slice-Prozessors hat die neue Maschine einen NS320032-Mikroprozessor.

Wirth taufte den Rechner Ceres. Im neuesten Informatik-Bulletin der ETH wird zwar behauptet, dies sei die Abkürzung für Computing Engine for Research, Engineering and Science. Doch in Klammern steht dabei, Ceres sei in der altitalischen Mythologie die Göttin der Fruchtbarkeit gewesen.

Der neue Ceres-Rechner ist in seiner Leistungsfähigkeit mit der VAX 11-780 von DEC vergleichbar. Was Wirth an der neuen Maschine gefällt, ist vor allem der niedrige Preis. Mit 11 500 Franken (rund 14 000 DM) lag der Stückpreis der Pilotserie von 30 Ceres-Maschinen erheblich unter dem, was auch beim gegenwärtig tiefen Dollar-Kurs für die aus den USA importierten Lilith-Rechner bezahlt werden muß.

Was Wirth an der Ceres weniger gefällt, ist die Leistungsfähigkeit des Mikroprozessors. Der NS32032 wurde nach aufwendiger Evaluation ausgewählt. Dabei verglichen Wirth und seine Mitarbeiter unter anderem die 32-bit-Mikroprozessoren von National Semiconductor und Motorola – also den NS32032 und den 68020 – mit dem alten, aus diskreten Bausteinen konstruierten Bit-Slice-Prozessor des Lilith. In seiner Beurteilung der beiden Mikroprozessoren stellt Wirth fest, die beiden Chips weisen zwar mit Abstand die besten Architekturen unter den kommerziell erhältlichen Produkten auf, doch merkt er nüchtern an, es sei „Raum für Verbesserungen vorhanden". Keiner der beiden Chips erreichte die Leistung des Lilith-Prozessors auch nur annähernd, jedoch der NS32032 war markant besser.

Modula-2 verbreitet sich

Modula-2 fand schnell Anhänger. „Pascal ohne die Nachteile von Pascal", pries Jerry Pournelle, der branchenbekannte Kolumnist des Magazins Byte, die neue Sprache. Vergangenes Jahr wurde Modula-2 zur Standardsprache an den beiden ETH von Zürich und Lausanne erklärt. Ihr Schöpfer erhielt 1984 die IEEE-Auszeichnung des amerikanischen Institute of Electrical and Electronic Engineering – so etwas wie den Nobelpreis der Branche. Zwei Jahre später folgte aus England der nach dem Erfinder des Konzepts aller digitalen Rechner benannte Turing-Award.

Von solchen Ehrungen macht Wirth kaum Aufhebens, er kann sich das nicht zuletzt deshalb leisten, weil er viele davon bekommen hat. Er brauchte nicht einmal den Finger zu heben, um Berufungen an die renommiertesten Forschungsinstitute und Laboratorien der Welt zu erhalten. Warum bleibt ein Mann wie Wirth in der Schweiz, wo er auf mangelnde Weitsicht, bürokratisches Beharrungsvermögen und nicht selten sogar Ignoranz trifft? Wirth sieht es so: „Mir gefällt es an der ETH. Ich habe die Möglichkeiten genutzt, etwas aufzubauen." Er baut immer noch auf. Gegenwärtig ist Niklaus Wirth dabei, ein portables Betriebssystem zu entwickeln.

Modulare Programmierung mit Modula-2

Jürg Gutknecht

Die tragende Idee der Modularisierung ist die Gliederung von Konstruktionen in autonome funktionale Einheiten, sogenannte Module. Im Gegensatz zu zahlreichen technischen Produkten wie Bauwerken, TV- und Hifi-Anlagen, Fotoapparaten, Computern und elektronischen Schaltungen entbehrten Software-Systeme bis vor kurzem einer expliziten modularen Struktur.

Das Konzept des Software-Moduls geht auf David Parnas zurück [1]. Parnas erkannte, daß zur Meisterung der Komplexität großer Programm-Systeme neue Methoden des Software-Engineerings erforderlich sind. Wenig überraschend schlugen sich die Parnas'schen Ideen alsbald in einer qualitativen Weiterentwicklung der Programmiersprachen nieder.

Tatsächlich ist in der Zwischenzeit eine Generation neuer, ,,modularer" Sprachen entstanden. Wir erwähnen in chronologischer Reihenfolge Mesa (Xerox PARC), Modula-2 [2], Chill und Ada. Die zweite dieser Sprachen, Modula-2, ist Gegenstand des vorliegenden Aufsatzes. Sie wurde von Niklaus Wirth definiert und trat um 1980 die Nachfolge von Pascal an.

Als bemerkenswertes Faktum ist zu erwähnen, daß Modula-2 aus Modula-1, einer Studie zur Programmierung gleichzeitig ablaufender Prozesse, hervorgegangen ist. In Modula-1 spielen die Module die Rolle sogenannter Monitoren [3], das sind Gruppierungen gemeinsamer Variablen und ihrer Operationen. Monitoren stellen kritische Programmabschnitte dar, die nicht gleichzeitig von verschiedenen Prozessen durchlaufen werden dürfen. Die viel wichtigere und allgemeinere Bedeutung des Modulkonzeptes für die Gliederung von Programmsystemen trat bei der Entwicklung erst später in den Vordergrund.

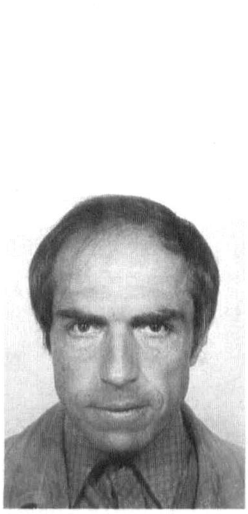
Wesentlich zum Verständnis modularer Systeme ist die Erkenntnis, daß die einzelnen Module zwar autonom, nicht aber unabhängig voneinander sind. Vielmehr sind sie in einem Netz von Abhängigkeiten verwoben. Der Konstrukteur eines Moduls wird seine Arbeit auf vorhandene, möglicherweise anderswo entwickelte Module abstützen. Entscheidend dabei ist, daß der Entwickler zwar deren Schnittstellen, also deren funktionale Definitionen genau kennen muß, nicht aber die Methoden ihrer Realisierung oder Implementation, zumindest nicht in allen Details.

Verweilen wir einen Moment bei diesem letzten Punkt. Er hat die weitreichende Konsequenz, daß keine direkten Abhängigkeiten zwischen Implementationen der verschiedenen Module bestehen. Nach der Festlegung der funktionalen Definitionen lassen sich in einem solchen System sämtliche Implementationen unabhängig voneinander ausarbeiten. Ohne das konstruktive Gebäude als Ganzes zu erschüttern, kann jede Implementation jederzeit durch eine neue ersetzt werden.

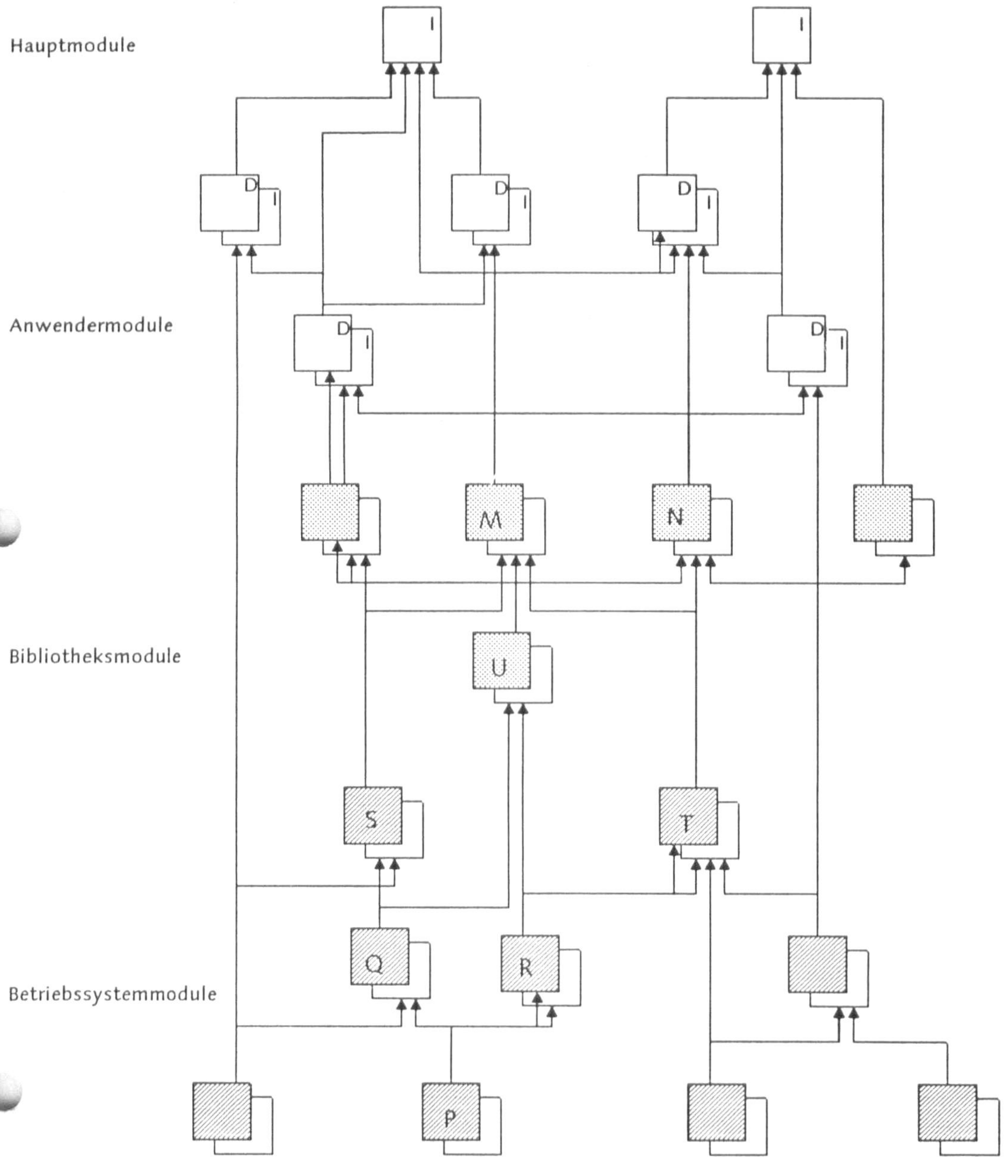
Betrachten wir nun das Software-System. Außer den beiden obersten Modulen besteht jedes Modul aus einem Definitionsteil und einem Implementationsteil. Auf den beiden obersten Modulen wird nicht weiter aufgebaut. Sie können als Hauptmodule interpretiert werden, die den dynamischen Programmablauf steuern, sobald ihnen die Kontrolle übergeben wird. Ein Hauptmodul, zusammen mit allen direkt und indirekt importierten, also den verwendeten Modulen, wird als Programm bezeichnet.

Wir bemerken, daß verschiedene Programme nicht notwendigerweise modulfremd sind. Beispielsweise werden Module M und N von beiden dargestellten Programmen importiert. Module, die so ausgelegt sind, daß sie in mehreren Programmen verwendet werden können, heißen Bibliotheksmodule. Bestimmte, in der Hierarchie weit unten angesiedelte Bibliotheksmodule stellen die Schnittstellen zu den Ressourcen des Computers wie Prozessor, Speicher, Disk, Netzwerk, Bildschirm, Eingabegerät usw. dar. In ihrer Gesamtheit bilden sie das Betriebssystem.

Da sich alle Module eines Programmsystems in einheitlicher Form präsentieren, wirkt ihre Einteilung in verschiedene Klassen künstlich. Tatsächlich tritt in modularen Software-Systemen die traditionelle Strukturierung in die horizontalen Schichten Betriebssystem, Bibliothek und Anwendung zugunsten einer thematischen Gliederung



Der Autor
Jürg Gutknecht ist Assistenzprofessor an der ETH Zürich. Nach der Ausbildung zum Systemprogrammierer bei der SWISS Air, studierte er Mathematik und Informatik an der ETH.



Schema eines modularen Software-Systems

in vertikale Programmstrukturen in den Hintergrund.

Modula-2

Eine wichtige Folge der Neuorientierung der Struktur modularer Systeme ist die Forderung nach einer Programmiersprache mit großer Einsatzbandbreite. Neben dem Angebot an Konstrukten zur Formulierung abstrakter Abläufe und Datenstrukturen ist die Möglichkeit der „maschinennahen" Programmierung von zentraler Bedeutung.

Der prinzipielle Aufbau der Sprache Modula-2, fortan kurz Modula genannt, stellt eine interessante Lösung dar, die sich das Modulkonzept selbst zunutze macht. Die eigentliche Sprache besteht aus einem minimalen Satz allgemeiner, maschinenunabhängiger Konstrukte. Alle systemabhängigen Elemente werden über Modulschnittstellen zur Verfügung gestellt. So beispielsweise Elemente zur Behandlung von Ein- und Ausgabe und von Files.

Das Standardmodul SYSTEM nimmt eine Sonderstellung ein. Es stellt die Verbindung der Sprache (bzw. des Compilers) mit dem Computer her und ermöglicht so die maschinennahe Programmierung. Beispielsweise läßt sich mit Hilfe dieses Moduls die Abbildung abstrakter Datenstrukturen in den Speicher explizit programmieren. Zur Illustration flechten wir an dieser Stelle Auszüge aus den Definitionsteilen der Module InOut und SYSTEM ein. Sie lassen erkennen, daß Operationen in Form von Prozedurköpfen definiert werden.

```
DEFINITION MODULE InOut;
FROM FileSystem IMPORT File;
```

VAR Done: BOOLEAN;
in, out: File;
Resultat der letzten Operation
Ein- und Ausgabefiles
PROCEDURE OpenInput (defext: ARRAY OF CHAR); Eröffne Eingabefile mit Namens-Suffix defext
PROCEDURE OpenOutput (defext: ARRAY OF CHAR); Eröffne Ausgabefile mit Suffix defext
PROCEDURE CloseInput;
PROCEDURE CloseOutput;
Schließe Eingabefile (Rückkehr zur Tastatur)
Schließe Ausgabefile (Rückkehr zu Bildschirm)
PROCEDURE Read (VAR ch: CHAR);
Lies nächstes Zeichen vom Eingabefile
PROCEDURE ReadString (VAR s: ARRAY OF CHAR); Lies Zeichenkette vom Eingabefile
PROCEDURE Write (ch: CHAR);
PROCEDURE Write Ln;
Schreibe nächstes Zeichen auf Ausgabefile
Schließe Zeile ab
PROCEDURE WriteString (s: ARRAY OF CHAR); Schreibe Zeichenkette auf Ausgabefile
END InOut;
DEFINITION MODULE SYSTEM;
TYPE ADDRESS = POINTER TO BYTE;
BYTE;
Für Motorola 68000
Adresse = Zeiger zu irgendeinem Byte
Kleinste adressierbare Einheit
PROCEDURE ADR (VAR x: Any Type): ADDRESS; Speicheradresse der Variablen x
PROCEDURE TSIZE (Any Type): INTEGER;
Größe in Byte von Any Type
PROCEDURE VAL (New Type; x: Any Type): New Type; Uminterpretation der Größe x als New Type
END SYSTEM;

Wie in Pascal ist in Modula jedem Objekt ein bestimmter, wohldefinierter Typ zugeordnet. Beispielsweise besitzt Done im Modul InOut den Standardtyp BOOLEAN und „in" den von FileSystem importierten Typ File. Operationen sind nur dann legal, wenn die Typen der beteiligten Objekte verträglich sind. Dynamische Datenstrukturen (z.B. Listen und Bäume) werden wie in Pascal mit typengebundenen Zeigern (POINTERN) verwirklicht.

Maschinennahe Programmierung

Die erwähnte Möglichkeit der maschinennahen Programmierung bei der Verwendung des Moduls SYSTEM beruht zu einem guten Teil auf den großzügigen Verträglichkeitsregeln der Typen BYTE und ADDRESS. ADRESS ist mit jedem Zeigertyp und mit ganzen Zahlen verträglich, BYTE mit jedem Typ von Bytegröße, und ARRAY OF BYTE mit jedem Typ überhaupt. Wir heben hervor, daß, im Gegensatz zu Zeigern, eine Variable vom Typ ADDRESS nicht notwendigerweise zur Basis eines Datenobjektes zeigen muß. Adressen können beliebige Speicherpositionen ansprechen. Die Verträglichkeit von Adressen mit ganzen Zahlen ermöglicht Adreßarithmetik.

Der Typ ARRAY OF BYTE gehört in die Kategorie der sogenannten dynamischen Arraytypen. Zusammen mit der Möglichkeit der maschinennahen Programmierung, den abstrakten Objekttypen und den Prozedurvariablen sind die dynamischen (oder offenen) Arrays die Grundpfeiler der mit dem Modulprinzip verknüpften Einrichtungen von Modula-2.

Dynamische Arrays

In Bibliotheksprozeduren, die einen Array-Parameter aufweisen, ist die Festlegung der Länge des Arrays oft unerwünscht. Der Kreis der potentiellen Kunden der Prozedur ist viel größer, wenn beim Aufruf ein Array beliebiger Länge akzeptiert wird. Dies kann in Modula dadurch erreicht werden, daß der Indexbereich in der Prozedurdeklaration offengelassen wird.
Beispiele:
PROCEDURE WriteString (s: ARRAY OF CHAR);
PROCEDURE InnerProduct (a, b: ARRAY OF REAL): REAL;
PROCEDURE WriteBlock (f: File; b: ARRAY OF BYTE);

WriteString schreibt eine Zeichenkette s beliebiger Länge auf den Bildschirm. InnerProduct ist eine Funktion. Sie akzeptiert Zahlenvektoren a und b beliebiger (gleicher) Länge und gibt deren Skalarprodukt als Resultat vom Typ REAL zurück. WriteBlock erweitert das File f um einen Block b beliebiger Größe und Struktur.

Abstrakte Objekttypen

Die dem Modulkonzept zugrundeliegende Idee der Datenkapselung ist dann in besonders reiner Form verwirklicht, wenn innerhalb des Moduls eine bestimmte Art von Objekten vollständig abgehandelt wird, d.h., wenn das Modul selbst sämtliche für diese Objektart benötigten Operationen zur Verfügung stellt. In diesem Fall ist die Struktur der Objekte für die Kunden des Moduls belanglos. Solche abstrakten Objekttypen

können in Definitionsteilen durch Offenlassen der Strukturangabe deklariert werden.
Beispiele:
TYPE BYTE;
TYPE Complex;
TYPE Viewer;

In den Beispielen werden der Reihe nach die Typen der Bytes, der komplexen Zahlen und der Bildschirmfenster abstrakt definiert.

Prozedurvariablen

Moderne interaktive Systeme sind gelegentlich „ereignisgesteuert". Dies bedeutet, daß das Ablauf-Steuerprogramm die „Ereignisse" (Eingabe, Zeigen auf ein am Bildschirm dargestelltes Objekt, Alarm der internen Uhr etc.) feststellt und den interessierten Programmen zur Behandlung übergibt. Dieser mit dem Hardware-Interrupt verwandte Mechanismus setzt die Möglichkeit der Registrierung von Behandlungsprozeduren voraus. In Modula lassen sich Prozeduren als Parameter übergeben und als Prozedurvariablen registrieren.
Beispiel:
TYPE Handler = PROCEDURE (Viewer, VAR Event);
PROCEDURE OpenViewer (VAR v: Viewer; h: Handler);

Viewer und Event sind Datentypen, die ein Bildschirmfenster, bzw. ein Ereignis beschreiben. Beim Aufruf von OpenViewer wird eine Behandlungsprozedur h übergeben. Diese wird als Prozedurvariable des Fensterobjektes v registriert und aufgerufen, wenn auf das betreffende Fenster gezeigt wird.

Damit kennen wir alle wichtigen modulorientierten Einrichtungen von Modula. Die Erfahrung hat gezeigt, daß ihre sinnvolle Verwendung, d.h. die gute Modularisierung, viel schwieriger ist, als man auf den ersten Blick vermutet. Der Modul-Designer sieht sich häufig mit kontroversen Zielsetzungen konfrontiert. Einige Grundregeln lassen sich aber dennoch angeben. Wir formulieren im nächsten Abschnitt drei der wichtigsten.

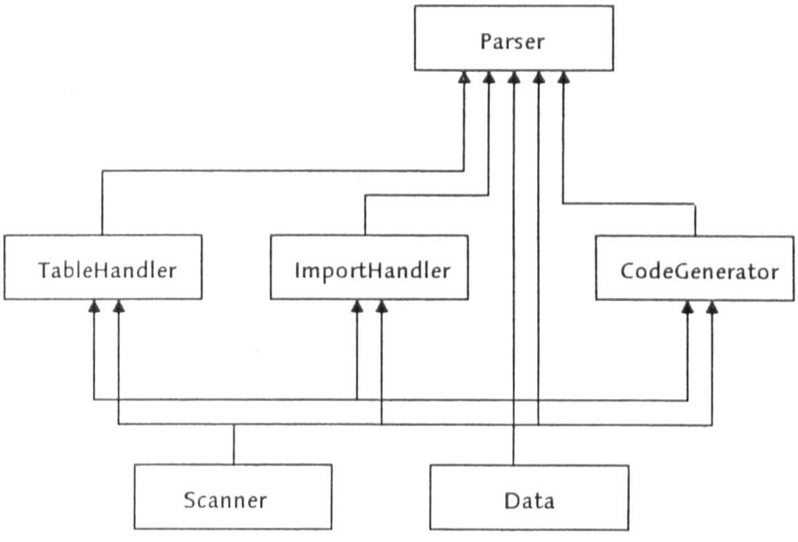
Richtlinien zum Design von modularen Systemen

Regel 1. Schnittstellenbreite

Die Breite der Modul-Schnittstellen, d.h. die Größe und Komplexität der Definitionsmodule ist ein Maß für die Qualität der Modularisierung. Je enger die Modul-Schnittstellen, desto natürlicher die Modularisierung.

Regel 2. Abhängigkeiten zwischen Modulen

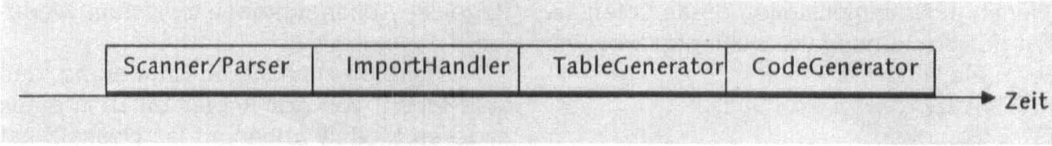
Wir rufen zunächst in Erinnerung, daß kein Modul von der Implementation eines anderen Moduls abhängig ist. Deshalb hat die Veränderung einer Implementation überhaupt keine Auswirkungen auf seine Umgebung. Hingegen invalidieren Modifikationen einer Modul-Definition prinzipiell alle direkt oder indirekt abhängigen Module. Beispielsweise macht die Modifikation der Definition von P die Module Q, R, T, U, M und N ungültig, da Q und R direkt und T, U, M und N indirekt von P abhängig sind. Der Modul S hingegen bleibt gültig. Unerwünschte Kettenreaktionen bei der Invalidierung von Modulen entstehen stets dann, wenn abhängige Modul-Definitionen (wie R und T) ins Spiel kommen. Deshalb sollte auf größtmögliche Unabhängigkeit der Modul-Definitionen geachtet werden.



Modularer Aufbau eines Einphasen Modula-Compilers

Eine nächste Beobachtung betrifft den gegenseitigen Import von Modulen. Drei der folgenden vier Szenarien sind zwar prinzipiell möglich, weisen jedoch unmißverständlich auf eine zu starke Verbindung zwischen den beteiligten Modulen hin und lassen die Auftrennung in separate Teile künstlich erscheinen.
Szenarium 1: (unmöglich)
DEFINITION MODULE A importiert DEFINITION MODULE B;
DEFINITION MODULE B importiert DEFINITION MODULE A.
Szenarium 2: (möglich)
DEFINITION MODULE A importiert DEFINITION MODULE B;
IMPLEMENTATION MODULE B importiert DEFINITION MODULE A.
Szenarium 3: (möglich)

Ablauf einer Meß-phasenübersetzung



IMPLEMENTATION MODULE A importiert DEFINITION MODULE B;
IMPLEMENTATION MODULE B importiert DEFINITION MODULE A.
Szenarium 4: (möglich)
IMPLEMENTATION MODULE A importiert DEFINITION MODULE B;
IMPLEMENTATION MODULE B importiert DEFINITION MODULE C;
IMPLEMENTATION MODULE C importiert DEFINITION MODULE A;

Regel 3. Granularität von Operationen

Die Granularität der Moduloperationen beeinflußt wesentlich den Kreis der potentiellen Anwender und damit den Erfolg eines Moduls. Sowohl zu komplexe als auch allzu einfache Operationen verringern die Attraktivität: Erstere, weil sie zu speziell sind, und letztere, weil sie nicht genügend Anreiz zu ihrer Verwendung geben.

Variante 3: Nachführen des Cursors entlang Gitterpunkten

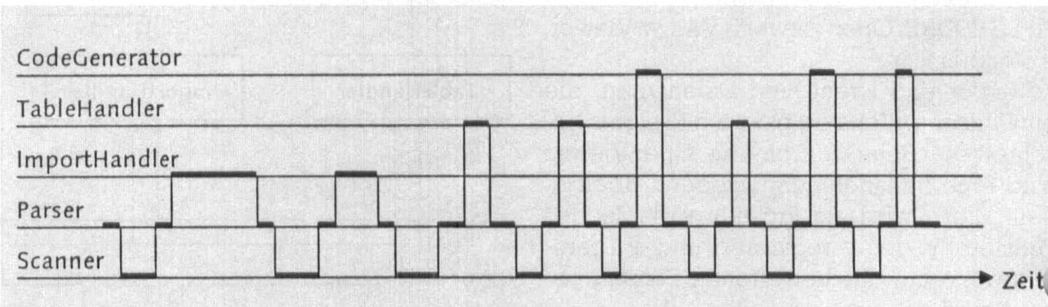
```
REPEAT GetMouse (buttons, x, y)
   DrawCursor (x DIV unit * unit, y DIV unit * unit); BusyRead (ch)
UNTIL (buttons { }) OR (ch # OC);
```

Variante 4: Verwendung eines privaten Cursormusters

```
REPEAT GetMouse (buttons, x, y);
   IF x < wid THEN SetPattern (private) ELSE SetPattern (standard) END;
   DrawCursor (x, y); BusyRead (ch)
UNTIL (buttons { }) OR (ch # OC);
```

Der Modula-Compiler als Beispiel zur Modularisierung

Wir illustrieren die Modularisierung anhand eines besonders wichtigen und attraktiven Programmes, nämlich eines Modula-Compilers. Der eigentlichen Behandlung der Modularisierungsfrage schicken



Das folgende Beispiel zur Behandlung des Maus-Zeigegerätes und Nachführen des Cursors am Bildschirm zeigt die Auflösung eines amorphen Blockes (TrackMouse) in atomare Operationen (GetMouse, DrawCursor, SetPattern, BusyRead) und die Möglichkeit zur Komposition von individuellen Mouse-Tracking Routinen.

Variante 1: Abbruch nach Loslassen aller Maustasten oder Eingabe eines Zeichens

```
REPEAT GetMouse (buttons, x, y);
   DrawCursor (x, y); BusyRead (ch)
UNTIL (buttons { }) OR (ch # OC);
```

Variante 2: Abbruch nach Loslassen der linken Maustaste oder Eingabe eines Zeichens

```
REPEAT GetMouse (buttons, x, y);
   DrawCursor (x,y); BusyRead (ch)
UNTIL NOT (left IN buttons) OR (ch # OC);
```

wir einige Bemerkungen über die Aufgabe und Funktionsweise des Compilers voraus, die für diesen Abschnitt von übergeordnetem Interesse sind.

Ein Software-System stellt eine Spezifikation von Abläufen dar. Damit diese Abläufe vom Computer ausgeführt werden können, muß das Software-System in der Maschinensprache des Computers vorliegen. Die Hauptaufgabe des Compilers ist die Übersetzung von Programmtexten in Folgen von Maschinenbefehlen.

Es wäre natürlich unpraktisch, wenn nach der geringsten Änderung das ganze System neu übersetzt werden müßte. Wir wissen beispielsweise bereits, daß sich eine Änderung innerhalb einer Implementation nicht auf die Umgebung auswirkt. Deshalb macht der Modula-Compiler Implementationsteile einer separaten Über-

setzung zugänglich. Während der separaten Übersetzung wird die Konsistenz mit dem zugehörigen Definitionsteil und mit den importierten Modulen überprüft. Diese Eigenschaft hebt die separate gegenüber der viel einfacher zu realisierenden unabhängigen Übersetzung entscheidend ab.

Nach dieser Beschreibung der Aufgabe des Compilers wenden wir uns seiner Konstruktion zu. Da der Modula-Compiler ein Programm ist, kommt sofort der Wunsch auf, ihn in seiner eigenen Sprache, also in Modula, zu formulieren. Dies wird noch verstärkt durch den folgenden Sachverhalt. Nach der Fertigstellung eines Compilers wird häufig der Wunsch laut, ihn einer neuen Zielmaschine anzupassen. Die Anpassung geht um so leichter von der Hand, je besser der zielmaschinenabhängige Teil isoliert ist. Im Idealfall besteht die Anpassung lediglich im Austausch eines Moduls.

Ein Einphasencompiler übersetzt Programmtexte in einem Durchlauf in Maschinenbefehle. Die Themen der Module entsprechen den Stationen, die bei der Verarbeitung der Texteinheiten zu Befehlsfolgen durchlaufen werden müssen.

Der Scanner übersetzt die Zeichenfolge in eine Folge von Modula-Sprachsymbolen. Der Parser prüft die syntaktische Korrektheit der Symbolfolge. Der TableHandler nimmt die deklarierten Objekte mit ihren Attributen in die sogenannte Symboltabelle auf. Der ImportHandler prüft die Konsistenz mit den importierten Modulen. Schließlich erzeugt der CodeGenerator die Folge der Maschinenbefehle.

Einige ergänzende Bemerkungen verschaffen dem Leser einen Blick hinter die Kulissen dieses Programmaufbaues. Zunächst erwähnen wir, daß der Parser gleichzeitig die Rolle des Dirigenten der Übersetzung einnimmt. Das Parser-Modul steuert als Hauptprogramm den dynamischen Ablauf.

Das bisher unerwähnte Modul Data zeigt eine interessante Spielart des Software-Moduls. Seine Aufgabe besteht nicht in der Ausführung irgendwelcher Operationen, sondern in der Präsentierung der gemeinsamen globalen Datentypen und Daten.

In jeder Phase findet bei einer sogenannten Mehrphasenübersetzung eine Transformation des Eingabestromes in einen Ausgabestrom statt. Der Eingabestrom der ersten Phase ist der Programmtext, der Ausgabestrom der letzten Phase ist die Folge der Maschinenbefehle.

Die Themen der Phasen entsprechen ziemlich genau den Themen der Module. Im Gegensatz zur statischen Gliederung des Programmes in Module handelt es sich hier um eine dynamische Gliederung in Phasen. Der fundamentale Unterschied zwischen den beiden Strukturierungsarten wird durch das Zeitdiagramm veranschaulicht.

Diesen Abschnitt abschließend weisen wir darauf hin, daß CodeGenerator das einzige wesentlich von der Zielmaschine abhängige Modul ist. Damit sind wir dem Ziel einer leichten Übertragbarkeit des Compilers auf neue Zielmaschinen ziemlich nahe gekommen.

Schlußfolgerung

Modula ist eine moderne Programmiersprache zur Konstruktion von großen modularen Software-Systemen. Als Strukturelement ist das Modul in seiner Relevanz durchaus mit der Prozedur und dem damit verbundenen Lokalitätsprinzip vergleichbar. Die separate Übersetzung ist gleichzeitig Garantie für effizienten Unterhalt und dauernde Konsistenz des Gesamtsystems.

Obwohl einfach in seiner Konzeption, hat sich das Modul als ein überraschend schwierig zu beherrschendes Hilfsmittel herausgestellt. Es tritt in vielfältiger Gestalt auf, z.B. als Package von Prozeduren, als Verwaltungsinstanz von Ressourcen, als Schnittstelle zur Hardware, als Black Box zur Verarbeitung von Objekten, als Monitor und als Datenbasis.

Das modulare Denken eröffnet neue Horizonte und führt zu weitreichenden und oft unerwarteten Konsequenzen, sogar für Betriebssystem-Designer und Computer-Architekten. Beispielsweise hat die Modulorganisation bereits ihren Niederschlag in neueren Mikroprozessor-Architekturen gefunden, so etwa in der Prozessor-Familie 32000 von National Semiconductor.

Stichworte:

Prozeß:
Ablauf einer logisch zusammengehörigen Folge von Aktionen

Monitor:
Aktuell im Zusammenhang mit Systemen, in welchen (quasi-)gleichzeitig mehrere Prozesse ablaufen. Monitoren sind Module, welche kritische Programmabschnitte enthalten, d.h. Abschnitte, die auf gemeinsamen Datenstrukturen operieren.

Literatur

[1] D.L. Parnas, *"On the criteria to be used in decomposing systems into modules." Communications of the ACM, 15, 1053–1058, Dezember 1972.*

[2] N. Wirth, *Programming in Modula-2, Springer Verlag 1985.*

[3] C.A.R. Hoare, *"Monitors: an operating system structuring concept." Communications of the ACM, 17, 549–557, Oktober 1974.*

Von Pascal zu Modula-2

Ulrich Kern

Modula-2 ist der Nachfolger von Pascal. Die Programmiersprache Pascal wurde ausschließlich zu Unterrichtszwecken erfunden. Aufgrund des Erfolges, den diese Sprache binnen kürzester Zeit in Informatikerkreisen genoß, wurde das ursprüngliche Lehrkonzept in vielerlei Hinsicht erweitert. Das hatte jedoch zur Folge, daß unabhängig voneinander mehrere, untereinander unverträgliche Pascal-Versionen entstanden. Alle Erweiterungen von Pascal hatten das Ziel, die konzeptionsbedingten Mängel zu entfernen.

Doch mit der Konkretisierung des Begriffs „Software-Engineering" stiegen auch die Ansprüche an eine moderne Programmiersprache: Sie sollte in jeder Beziehung modular sein – bei der Planung großer Software-Projekte werden die Modulschnittstellen definiert, die einzelnen Module werden unabhängig davon realisiert – und sie sollte eine Universalsprache sein – Werkzeuge zur Hardware-nahen Programmierung sind ebenso vorhanden wie beliebig abstrakte Datentypen. Diese Kriterien wurden von den einzelnen Pascal-Dialekten nur in unzureichender Weise erfüllt. So war es nur folgerichtig, daß, aufbauend auf dem erfolgreichen Konzept von Pascal, neue, große Programmiersprachen geschaffen wurden: Ada und Modula-2.

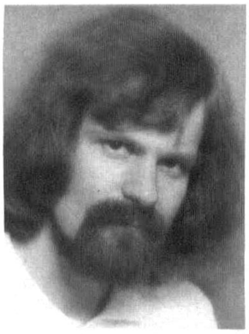
Während die vom Verteidigungsministerium der USA initiierte Programmiersprache Ada aufgrund ihres Umfangs nur auf Großrechnern implementiert werden kann, liegen inzwischen Modula-2-Compiler für nahezu alle Rechnergattungen vor, bis hinunter zu MS-DOS und CP/M-Maschinen.

Inzwischen ist eine beinahe unübersehbare Menge hervorragender Software in Pascal geschrieben worden, und es stellt sich nun die Frage, wie die in dieser Sprache formulierten Programme nach Modula-2 übertragen werden können. Da beide Sprachen aus einer Quelle, nämlich vom Schweizer Professor Niklaus Wirth, kommen, ist eine solche Übertragung ohne größere Probleme möglich. Das wichtigste Werkzeug hierfür ist ein leistungsfähiger Textprozessor. Die folgenden Hinweise sind sicherlich keine vollständige Übersetzungsanleitung. Sie sollen vielmehr als Hilfestellung dienen bei einer der unbeliebtesten und zeitraubendsten Arbeit am Computer.

Am leichtesten sind solche Änderungen durchzuführen, bei denen einzelne Sprachkonstrukte nur umbenannt wurden. So ist das Pascal-Schlüsselwort PROGRAM einfach durch das Modula-2 Äquivalent MODULE zu ersetzen. Eine eventuell dem Programmnamen folgende Dateiliste muß weggelassen werden. Wichtig in diesem Zusammenhang ist die Tatsache, daß in Modula-2 Groß-/Kleinschreibung unterschieden wird. Schlüsselwörter und Standardbezeichner werden in Modula-2 grundsätzlich großgeschrieben. Bei Modula-2-Implementationen auf Multitask- und Mehrplatzsystem kann dem Programmnamen noch eine Prioritätsangabe folgen. Einzelheiten hierzu sind aus den Systemunterlagen ersichtlich.

In Modula-2 beginnen Kommentare ausschließlich mit "(*" und enden mit "*)". Die geschweiften Klammern sind für andere Zwecke reserviert. Allerdings können Kommentare beliebig verschachtelt werden, so daß hier eine einfache, globale Textersetzung ganze Arbeit leistet.

Ein weiteres Schlüsselwort von Pascal kann ebenfalls relativ einfach durch das entsprechende Modula-2-Wort ersetzt werden: FUNCTION wird zu PROCEDURE. Hier gilt es allerdings die Besonderheit zu beachten, daß eine Funktions-Prozedur (so heißen die Funktionen in Modula-2) immer eine Parameterliste haben muß, die jedoch auch leer sein kann. Diese Änderung ist bei den jeweiligen Funktionsaufrufen zu berücksichtigen. Der Funktionswert wird in Modula-2 nicht mittels einer Zuweisung an den Funktionsnamen festgelegt, sondern durch das neue Schlüsselwort Return. Zudem muß bei allen Prozeduren am Blockende der Prozedurname wiederholt werden – ein Zwang, dem sich viele Pascal-Programmierer schon lange dadurch freiwillig unterzogen haben, daß sie den Namen am Blockende als Kommentar nochmals aufführten.



Der Autor
Ulrich Kern studierte Logik und Wissenschaftstheorie in München. Er erteilt Programmierkurse und ist der Autor der CHIP-SPECIAL-Reihe „Turbo-Pascal".

Pascal:

```
FUNCTION Fakultaet(n:INTEGER):INTEGER;
  BEGIN
    IF n=0 THEN Fakultaet:=1
           ELSE Fakultaet:=n*Fakultaet(n-1)
  END;
```

Modula-2:

```
PROCEDURE Fakultaet(n:INTEGER):INTEGER;
  BEGIN
    IF n=0 THEN RETURN 1
           ELSE RETURN n*Fakultaet(n-1)
    END (* IF-Ende, siehe weiter unten *)
  END Fakultaet;
```

Ebenfalls wenig Schwierigkeiten macht eine sprachkonzeptionelle Veränderung: Die zusammengesetzte Anweisung von Pascal (BEGIN Anweisungssequenz END) gibt es in Modula-2 nicht mehr. Hier hat jede strukturierte Anweisung (mit Ausnahme von REPEAT Anweisungssequenz UNTIL . . .) ein eigenes END.

Dazu ein Beispiel:

Pascal:

```
IF BoolscherAusdruck
  THEN BEGIN Anweisungssequenz END
  ELSE BEGIN Anweisungssequenz END
```

Modula-2:

```
IF BoolscherAusdruck
THEN Anweisungssequenz
ELSE Anweisungssequenz
END
```

Auf diese Abwandlung ist insbesondere dann zu achten, wenn auf THEN oder ELSE nur eine einfache Anweisung folgt. Auch dort muß END die bedingte Anweisung abschließen. Eine Unterlassung kann vom Compiler erst am Ende des zugehörigen Blockes erkannt, der Fehler somit schwer lokalisiert werden.

Dasselbe gilt für die WHILE-Anweisung und die FOR-TO-Schleife. Das Schlüsselwort DOWNTO gibt es in Modula-2 nicht mehr. Hier kann die Schrittweite mit dem Wort BY festgelegt werden.

Somit wird aus

```
FOR Variable := Ausdruck DOWNTO Ausdruck DO
```

die Modula-2-Konstruktion

```
FOR Variable := Ausdruck TO Ausdruck BY -1 DO.
```

Das Herunterzählen mit – 1 funktioniert bei allen Aufzählungstypen, also nicht nur bei INTEGER.

Das neue Konzept der Anweisungssequenz findet auch in der CASE-Anweisung Verwendung. Allerdings wird nur die gesamte Anweisung mit END abgeschlossen, das Ende der einzelnen Anweisungssequenzen werden durch das Zeichen „|" kenntlich gemacht.

Pascal:

```
CASE Ausdruck OF
   Konst : BEGIN Anweisungssequenz END;
   Konst : Anweisung
END (* CASE *)
```

Modula-2:

```
CASE Ausdruck OF
   Konst : Anweisungssequenz |
   Konst : Anweisung
END (* CASE *)
```

Das Schlüsselwort PACKED ist in Modula-2 nicht mehr vorhanden, es wird einfach ersatzlos gestrichen.

Etwas schmerzlicher wird von manchen Programmierern das Fehlen von GOTO empfunden. Andere wiederum bejubeln gerade frenetisch den Wegfall des letzten Relikts aus unstrukturierten Programmierzeiten. Wie dem auch sei, sollen Pascal-Programme übertragen werden, so müssen die GOTO-Anweisungen eliminiert werden. Ein häufiger und auch von Struktur-Experten gern gesehener Gebrauch ist ein Sprung an das Ende Prozedur. Befindet sich das angesprungene Label unmittelbar vor dem blockabschließenden END, so kann anstelle der „GOTO"- die RETURN-Anweisung eingesetzt werden. Wurde der Sprung dazu genutzt, eine Schleife vorzeitig zu verlassen, so kann hier die „LOOP"-Anweisung in Verbindung mit EXIT angewandt werden. Mit LOOP wird eine Endlosschleife programmiert, die an beliebigen Stellen verlassen werden kann. In den meisten anderen Fällen kann eine (unter Umständen mehrfach verschachtelte) bedingte Anweisung den Sprung überflüssig machen. Modula-2 bietet, auch für solche Fälle, eine Erweiterung der bedingten Anweisung, mit deren Hilfe solche Anweisungen übersichtlicher und einfacher werden:

```
IF BoolscherAusdruck THEN Anweisungssequenz
ELSIF BoolscherAusdruck THEN Anweisungssequenz
ELSIF  ...
ELSE Anweisungssequenz END
```

Das Schlüsselwort ELSIF erlaubt eine mehrfache Abfrage, ohne die Verschachtelungstiefe zu erhöhen.

Alle anderen Sprünge, insbesondere wenn sie über Blockgrenzen hinausgehen, können erst aufgrund einer genauen Programmanalyse ersetzt werden. Einzige Ausnahme ist ein Sprung ans Programmende. An dessen Stelle tritt die Standardprozedur HALT.

Bei der Typ- und Variablendeklaration ist darauf zu achten, daß Unterbereiche in eckigen Klammern angegeben werden. Bei der Deklaration von Feldern (ARRAY) werden, wenn es sich bei dem Index nicht um Unterbereiche handelt, die eckigen Klammern weggelassen.

Pascal:

```
TYPE Byte = 0..255;
     SpeicherBlock = ARRAY[Byte] OF Byte;
     ZeichenTabelle = ARRAY[CHAR] OF CHAR;
     Vektor = ARRAY[1..3] OF REAL;
```

Modula-2:

```
TYPE Byte = [0..255];
     SpeicherBlock = ARRAY Byte OF Byte;
     ZeichenTabelle = ARRAY CHAR OF CHAR;
     Vektor = ARRAY[1..3] OF REAL;
```

Schwieriger wird die Übersetzung, wenn in Pascal-Programmen Datentypen benutzt werden, die es in Modula-2 nicht mehr gibt. Das sind „TEXT" und „FILE OF", sowie, mit Einschränkung, „SET OF". Der Wegfall der (Text-)Dateien beruht auf der vollkommenen Auslagerung aller Ein-/Ausgaberoutinen. Bei Mengen, die nach wie vor vorhanden sind, ist die Anzahl der Elemente auf die Wortbreite des Prozessors beschränkt, auf dem die Modula-2-Implementation läuft, also üblicherweise 16 oder 32. Für Mengen, die auf selbstdefinierten Aufzählungstypen basieren, reicht diese Anzahl meistens aus. Eine Konstruktion wie SET OF CHARACTER ist allerdings nicht möglich. Damit an solch großen Mengen eine Übertragung nicht scheitert, gehört zu den meisten Modula-2-Systemen ein externes Modul namens LONGSETS, das den fehlenden Datentyp mit allen benötigten Prozeduren bereitstellt. Ein Einsatz dieses Moduls setzt eine gewisse Vertrautheit mit dem Definitionsteil voraus.

Zwei weitere Besonderheiten müssen beim Einsatz von Mengen noch berücksichtigt werden: Mengenkonstanten werden nicht in eckigen, sondern in geschweiften Klammern angegeben. Bei der Angabe einer Mengenkonstanten dürfen keine Variablen verwendet werden. Dafür gibt es die beiden Standardprozeduren „INCL" und „EXCL" zum Ein- bzw. Ausschließen von Mengenelementen.

Die meiste Arbeit schließlich macht die Umwandlung der Datenein- und -ausgabe. Bei Modula-2 wurde das gesamte IO-Handling in sogenannte Basismodule ausgelagert, aus denen die benötigten Prozeduren erst importiert werden müssen:

```
FROM Modulname IMPORT  Bezeichner1,
  Bezeichner2, ... ;
```

Diese Import-Listen kommen noch vor dem Deklarationsteil eines Programms. Für Tastatureingabe und Bildschirmausgabe ist das Modul „InOut" zuständig. Es liefert so wichtige Prozeduren wie

```
Read          ein Zeichen (CHAR) lesen
ReadString    eine Zeichenkette lesen
ReadInt       eine INTEGER-Variable
              lesen
WriteLn       Zeilenvorschub
Write         ein Zeichen schreiben
WriteString   eine Zeichenkette
              schreiben
WriteInt      eine INTEGER-Variable
              schreiben
```

Da alle diese Prozeduren immer nur einen Wert lesen beziehungsweise schreiben können, müssen die entsprechenden Pascal-Anweisungen meist aufgeteilt werden:

Pascal:

```
WriteLn('Die Summe ist ',A+B:5);
```

Modula-2:

```
WriteString('Die Summe ist ');
WriteInt(A+B,5);
WriteLn;
```

Werden in Pascal-Programm Dateien vom Typ TEXT benötigt, so gibt es in Modula-2 zwei Möglichkeiten. Wird nur kurzzeitig die Ein- oder Ausgabe umgeleitet, so liefert das schon erwähnte Modul InOut die Prozeduren OpenInput, CloseInput, OpenOutput und CloseOutput sowie die boolsche Variable „Done". Die Arbeitsweise soll am Beispiel der Eingabeumlenkung verdeutlicht werden. Der Aufruf von OpenInput bewirkt, daß während des Programmlaufs an dieser Stelle vom Anwender ein Dateiname erfragt wird. Diese Datei wird, wenn möglich, eröffnet. Im Anschluß daran beziehen sich alle Leseanweisungen, die ebenfalls aus InOut importiert wurden, auf

diese Datei. Nach dem Aufruf von CloseInput wird ganz automatisch wieder die Konsole zum Standardeingabegerät. Die Variable „Done" gibt nach jeder Operation Auskunft über den Erfolg.

Die zweite Möglichkeit liegt im Import des Datentyps TEXT und aller benötigten Prozeduren aus dem Basismodul Texts. Dabei kann es vorkommen, daß mehrere gleichnamige Prozeduren importiert werden müssen, Read aus InOut und Read aus Texts. Auch das ist möglich. Dazu wird der sogenannte qualifizierende Import genutzt:

```
IMPORT Texts;
IMPORT InOut;
```

Dadurch werden alle Bezeichner, die dieses Modul exportiert, dem Programm verfügbar gemacht. Es muß nur beim Zugriff der Modulname mit angegeben werden.

Texts.Read(...

InOut.Read(...

Für die Bearbeitung anderer Dateien ist das Modul FILES zuständig. Aus diesem können der Datentyp FILE sowie die benötigten Prozeduren importiert werden. Bei einem Einsatz muß auf jeden Fall die Beschreibung des Moduls konsultiert werden.

Zum Glück hat sich an dem Zeigerkonzept von Pascal nicht viel geändert. Anstelle des Hochpfeils in Deklarationen heißt es in Modula-2 POINTER TO. Die Standardprozeduren NEW und DISPOSE sind zwar weiterhin verfügbar. Da sie jedoch auf die Lowlevel-Routinen ALLOCATE und DEALLOCATE zurückgreifen, müssen diese aus dem Modul Storage importiert werden.

Manchmal kommt es vor, daß in Pascal-Programmen Standardprozeduren oder -funktionen umdefiniert werden, obwohl es kaum einen hinreichenden Grund für dieses Unterfangen gibt. Da in Pascal die Standardbezeichner als in einem äußeren Block definiert angesehen wurden, war dies auch ohne weiteres möglich. In Modula-2 werden Standardbezeichner als automatisch importiert betrachtet. Aus diesem Grund dürfen sie auf Modulebene nicht umdefiniert werden, sondern nur innerhalb von Prozeduren.

Bei dem Datentyp RECORD mit Varianten hat sich die Syntax geändert. Hier müssen die runden Klammern weggelassen werden. Die einzelnen Varianten werden, wie bei der CASE-Anweisung, durch "|" kenntlich gemacht. Da in Modula-2 mehrere variante Teile vorkommen dürfen, muß jeder mit END abgeschlossen werden. Falls das Auswahlfeld leer bleibt, muß trotzdem der Doppelpunkt geschrieben werden.

Pascal:

```
Person = RECORD
          Name, Vorname : ARRAY[0..20] OF CHAR;
          CASE Stand : (verh, gesch, ledig) OF
           verh, gesch : (Kinderzahl : INTEGER);
           ledig : (Steuerklasse : 1..6)
         END (* CASE und RECORD *);
```

Modula-2:

```
Person = RECORD
          Name, Vorname : ARRAY[0..20] OF CHAR;
          CASE Stand : (verh, gesch, ledig) OF
            verh, gesch : Kinderzahl : INTEGER |
            ledig : Steuerklasse : [1..6]
          END (* CASE *)
         END (* RECORD *);
```

Allerdings werden in Modula-2 trickreiche Konstruktionen zur Typaufhebung nicht mehr benötigt, da hierfür das Konzept des Re-Typings zur Verfügung steht.

Ein Feature, das leider nur sehr wenige Pascal-Versionen bieten, ist die Möglichkeit, Prozeduren und Funktionen selbst als Parameter zu übergeben. In Modula-2 können beliebige Prozeduren als Datentyp deklariert werden. Somit gibt es auch Variable und Parameter von solchen Prozedur-Typen. Ein häufig anzutreffendes Beispiel ist die Integration reeller Funktionen:

```
TYPE Funktion = PROCEDURE(REAL):REAL;
VAR f : Funktion;

PROCEDURE Integral(f : Funktion; von,bis : REAL):REAL;
  ...
  END Integral;
  ...
  f:=sin; WriteReal(Integral(f,0,1),10);
 (* oder WriteReal(Integral(sin,0,1),10) *)
```

Zuletzt soll noch der Ersatz von zwei wichtigen Pascal-Standardfunktionen besprochen werden: SUCC und PRED gibt es nicht mehr. An ihre Stelle traten die Prozeduren INC und DEC. Damit wird das beliebte Pascal-Konstrukt "i:=SUCC(i)" zu "INC(i)".

Für einige Pascal-Versionen wie bspw. Turbo-Pascal gibt es inzwischen schon Übersetzungsprogramme, die einen Großteil der Arbeit selbstständig erledigen. Diese Programme greifen auf mitgelieferte Module zu, in denen die Dialekt-Besonderheiten implementiert sind. Der Rest sollte mit Hilfe der hier vorgestellten Tips kein Problem sein.

Warum Farsight in Modula-2 programmiert wurde

Albert Meier

Beim Start von Farsight, Anfang 1984, stand die Idee, ein integriertes Software-Paket für IBM PC und kompatible Rechner zu entwickeln. Es sollte die Leitung von Einzelkomponenten aufweisen und für beliebige Erweiterungen offenstehen.

Mancher Leser kann sich noch an diese Zeit erinnern: Der IBM PC war noch nicht lange vorgestellt und es war noch sehr zweifelhaft, ob er sich durchsetzen würde. Mit aller Deutlichkeit zeigte sich, daß bestehende Software, vor allem vom älternen Betriebssystem CP/M-80, nicht einfach so auf das neue PC-DOS des IBM PC übertragen werden konnte, weil die neue Hardware neue Software-Konzepte verlangte um ganz ausgeschöpft zu werden.

Gerade zu dieser Zeit machte ein völlig neues Paket Furore, welches in der Zwischenzeit die Software-Welt erobert hat: 1-2-3 von Lotus. Im Grunde war die Idee nicht so neu; VisiCalc, welches einige Jahre vorher auf den Markt gekommen war, bot praktisch dasselbe, und doch zeigte sich gerade da, daß Ideen auf dem IBM PC anders realisiert werden konnten und mußten als auf den älteren Computersystemen. Viele Benutzer fanden die Übertragung alter Programme auf das neue Betriebssystem wenig attraktiv. Befehle und Möglichkeiten standen bei diesen Programmen zwar mehr als genug zur Verfügung, auch die Übertragung auf das PC-DOS war meist nicht schlecht gelungen, doch war es den meisten Benutzern zu unverständlich, zur Bedienung Hunderte von Befehlen zu lernen, um die Fähigkeiten der Programme ausnutzen zu können. Zu dieser Zeit gab es aber noch ein paar andere Computer, die den IBM PC und seine Programme wie aus der Steinzeit ausschauen ließen, allen voran die Apple-Computer Lisa und Macintosh, die sich durch eine elegante Benutzerführung auszeichneten, die in den Entwicklungslaboratorien von Xerox in Palo Alto entwickelt wurden und die Dokumente in vielen Fenstern auf dem Bildschirm anzeigte, die sich öffnen und schließen, vergrößern und verkleinern ließen. Dazu waren die Befehle in kleinen Menüs versteckt, die auf den Bildschirm „herausgezogen" werden konnten. Somit entfiel das mühselige Suchen von Befehlen in der Dokumentation, und die Dokumente sahen so aus, wie man es vom Schreibtisch her gewohnt war. Solches konnten die alten CP/M-Programme

```
 Pult  Datei  Arbeit Bereich Transfer Output  Kartei
-C:\TEXT\AMORTISA.WKS-----------  C:\TEXT\ITALIENI\*.*
C9: @PMT(D2,D3/12,60)
                                 <Zurück auf das Verzeichnis TEXT>
       A          B              APPENDIA WRD      3124   3-07-87   15:37:20
7    Monat  Investition  Zah     APPENDIB WRD      4692   3-07-87   15:38:10
8                                APPENDIC WRD      3222   3-07-87   15:38:26
9          1   12,500.00         APPENDID WRD      3218   3-07-87   15:39:08
10         2   12,338.58         APPENDIE WRD      4044   3-07-87   15:39:19
-C:\TEXT\ARTIKEL\FSMODULA.WRD    APPENDIF WRD     25297   3-07-87   15:42:05
B|S. 14    Z. 9     Col 25  S    CAP1     WRD     10103   3-07-87   15:42:29
                                 CAP2     WRD     10276   3-07-87   15:43:26
          A.+ L. Meier-Vogt      CAP3     WRD     62606   3-07-87   15:49:02
          in loser Folge ein     SCRIPT-I SCR        38   3-07-87   15:37:02
          Informationsschrif     10 Dateien    126620    16353280 verfügbar
          bezogen werden kan
          verschiedene Dienste auf Mailboxen und auch       4   10,823.89
          auf Public-Domain-und Demonstrationsdisketten     9   10,648.50
          angeboten. A.+ L. Meier-Vogt führt neben M2SDS    5   10,471.65
          für IBM PC auch verschiedene andere Modula-2      2   10,293.32
          Software-Entwicklungssysteme, z.B. für            A:
          Commodore Amiga (am unteren Ende) oder für
          IBM/370-Mainframes (am oberen Ende). Die Firma
          ist weltweit aktiv und besitzt Vertretungen in
```

nicht bieten, und auch 1-2-3 reichte bei weitem nicht an einen solchen Bedienungskomfort heran. Trotzdem verkauften sich die IBM PC und damit die dazugehörige Software besser als die Apple-Computer.

Was war zu tun, wenn ein neues Softwarepaket entwickelt werden sollte, das sowohl eine moderne, komfortable Umgebung mitbrachte, damit sich jeder Anwender wie mit dem Apple Macintosh sofort vertraut fühlen konnte, dazu in Leistung und Fähigkeiten den älteren und neueren Programmen mindestens ebenbürtig war und – das war die Hauptschwierigkeit – auf dem IBM PC in der Standardausbaustufe sein sollte.

Versuche von Firmen, die Apple-ähnliche Window-Umgebungen entwickelten, scheiterten regelmäßig an der zu geringen Leistungsfähigkeit und an der komplizierten Programmierung. Andere Firmen brachten mit etwas mehr Erfolg sogenannte integrierte Pakete auf den Markt, die versuchten, möglichst viele Funktionen in einem einzigen Paket zu vereinen und damit die Anzahl der Befehle zu verringern. In den verschiedenen Programmteilen können dieselben Befehle für gleiche oder ähnliche Funktionen verwendet werden. Solche Programme zeichneten sich vor allem durch einen hohen Speicherbedarf und dicke Handbücher aus, da nach dem Motto „alles oder nichts" immer alle Programmteile zur Verfügung stehen müssen, sollte die wirkliche Integration gewährleistet sein. Außerdem waren integrierte Pakete sehr teuer.

Interface Technologies Corp., einem Software-Unternehmen in Houston, Texas, und deren europäischem Partner Meier-Vogt in Bonstetten, Schweiz, schwebte ein anderes Konzept vor. Um einen sehr kleinen, klar definierten Kern, den Window-Manager, sollten die verschiedenen Programmteile der integrierten Software so gruppiert werden, daß vom Computer her betrachtet jeder Teil als eigenständiges unabhängiges Programm funktioniert. Der Window-Manager sollte nur das darunterliegende Betriebssystem über Fenster für den Benutzer abbilden und die Verbindung zwischen den einzelnen Komponenten sicherstellen. Daß das neue integrierte Paket „Farsight" heißen sollte, war von Anfang an klar, doch mit welcher Programmiersprache die hohen Anforderungen erfüllt werden konnten, war noch nicht entschieden.

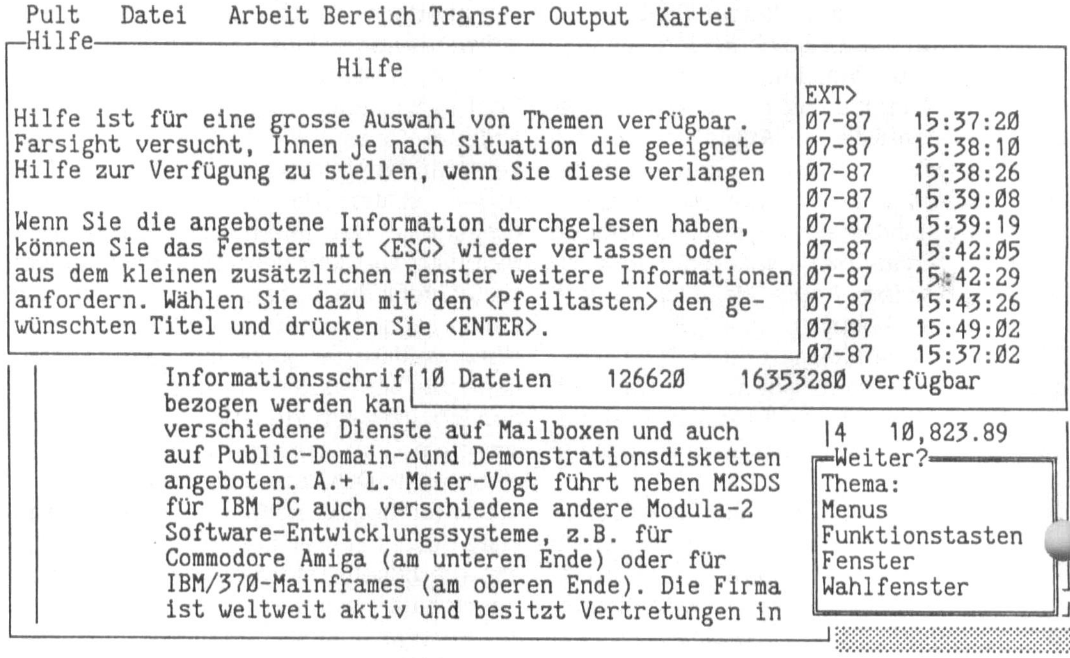
Farsight mußte mehrere Arbeiten gleichzeitig, also im Hintergrund durchführen können, da sonst die nötige Geschwindigkeit nicht erreicht werden konnte. Der IBM PC war zu langsam, um nach jedem Tastendruck in der Textverarbeitung die Formatierung des bearbeiteten Texts auf dem Bildschirm schnell genug zu überprüfen und gleichzeitig den Tastatur-Buffer zu leeren. Diese Arbeit mußte in den Hintergrund verlagert werden, genauso wie auch die Ausgabe auf bis zu zwei Druckern.

Es galt, eine Programmiersprache zu verwenden, die in allen Bereichen extrem schnell arbeitete. Es waren Fenster zu öffnen und zu schließen, den Speicher virtuell

```
  Pult   Datei   Arbeit Bereich Transfer Output  Kartei            OK
 C:\TEXT\AMORTISA.WKS
 C9: @PMT(D2,D3/12,60)

          A          B          C            D           E              :16:01
 7      Monat  Investition  Zahlungen      Zins     Amortisation        :23:23
 8                                                                      :23:25
 9          1   12,500.00     265.59      104.17         161.42         :28:17
 10         2   12,338.58     265.59      102.82         162.77         :39:05
 11         3   12,175.81     265.59      101.47         164.12         :16:02
 12         4   12,011.69     265.59      100.10         165.49         :23:29
 13         5   11,846.20     265.59       98.72         166.87         :05:22
 14         6   11,679.33     265.59       97.33         168.26         :07:08
 15         7   11,511.07     265.59       95.93         169.66         :51:08

                                                              A:
```



zu verwalten und dabei nicht langsamer zu sein als vergleichbare selbständige Programme. Die virtuelle Speicherverwaltung war übrigens aus dem Grund nötig, daß der IBM PC einen Speicher von maximal 640 KByte verwalten konnte und dazumal normalerweise nicht mehr als 256 KByte Speicher in der Grundausstattung besaß. Es ging also darum, dem Benutzer möglichst zu verbergen, daß nur ein beschränkter Speicherplatz zur Verfügung stand, ohne daß er dies an einem Geschwindigkeitsverlust merken sollte. Insgesamt gesehen mußte einfach alles schnell gehen!

Die Informatiker der beiden Firmen waren sich außerdem einig, daß sie eine wartbare Software entwickeln wollten, also ein Computerprogramm, welches auch nach vielen Jahren noch verbessert werden kann, ohne daß man es neu schreiben muß. Die verwendete Notation (oder Programmiersprache, wie man üblicherweise sagt) mußte also jederzeit und auch von neu hinzugekommenen Mitarbeitern leicht gelesen werden können, Klarheit und Aussagekraft garantieren.

Da alle Komponenten von Farsight voneinander unabhängig sein mußten und zu deren Erstellung auch verschiedene Teams an verschiedenen Orten eingesetzt wurden, drängte sich eine modulare Programmierung auf; vor allem kam es darauf an, daß die Schnittstellen zwischen den einzelnen Programmteilen wie auch zwischen den Teams klar definiert waren. Da man annehmen konnte, daß die Computerwelt nicht beim IBM PC stehenbleibt, sondern sich lebendig weiterentwickelt und es ja auch noch andere Computer gab und gibt, war es wichtig, möglichst wenig ändern zu müssen, wenn es galt, die Programme auf einen anderen Computer oder ein anderes Betriebssystem zu übertragen.

Zur Entwicklung von Farsight wurde Modula-2 eingesetzt. Es bietet leicht kontrollierbare Mechanismen für parallele Prozesse, ohne daß das darunterliegende Betriebssystem ebenfalls gleichzeitig mehrere Aufgaben bewältigen können muß oder auf mehr als einen Prozessor zugreifen kann. Da Modula-2 mit sehr wenig Elementen auskommt, ist es außerordentlich leicht zu übertragen und bietet, quasi als Nebeneffekt, eine hohe Geschwindigkeit, die nur von der Programmierung in Maschinensprache übertroffen wird.

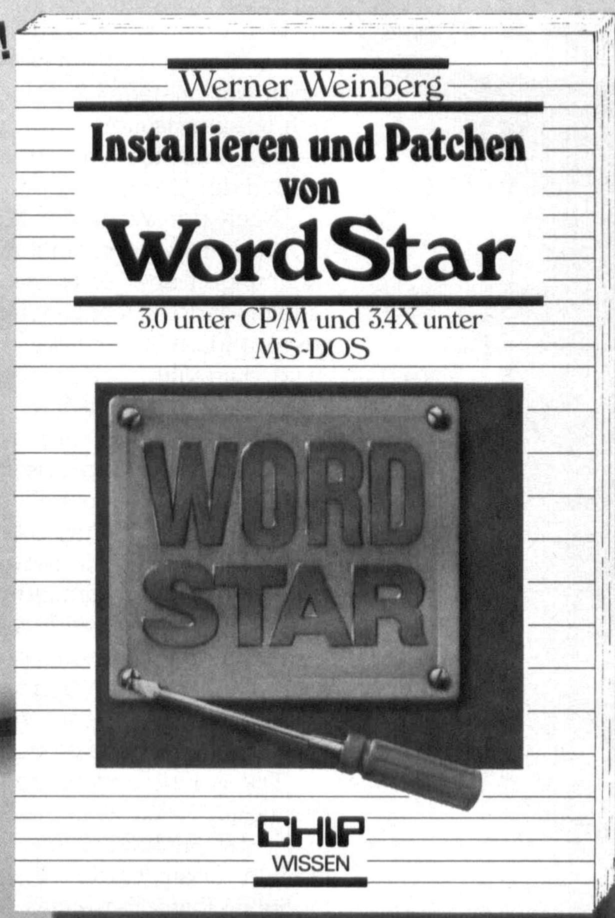
Wie der Name Modula-2 schon sagt, werden Programme in Moduln aufgebaut. Im Definitionsmodul wird die Schnittstelle festgelegt. Das Implementationsmodul enthält die „Beschreibung", wie das Modul sich verhält. Auf diese Weise muß jedes Team nur die Definitionsmodule des anderen kennen und im Gegenzug die eigenen bekanntgeben, damit die beiden Arbeiten zusammenpassen. Gerade für Farsight hat Modula-2 auch den Vorteil, daß somit eine wirkliche Trennung zwischen den einzelnen Programmteilen vorliegt, was sie voneinander unabhängig macht. Jeder Programmteil kann also auf den Window-Manager zugreifen ohne Ballast eines ande-

CHIP WISSEN

Neuerscheinung

Wichtige Bücher für die PC-Praxis!





Rainer Schwinn

dBASE III PLUS kurz und bündig

Das Buch vermittelt in methodisch aufeinander aufbauenden Abschnitten die Abfragesprache dieses Datenbanksystems. Der Leser lernt, wie er mit der hervorragenden Benutzerführung des Systems arbeitet, wie er Dateien erstellt, verändert und mit wenigen Befehlen auswertet, wie er Dateien sortiert, indiziert und in einer gewünschten Form auf Bildschirm oder Drucker ausgibt. Er erfährt, wie er mit dem integrierten Programmgenerator eigene Anwenderprogramme zur Lösung seiner EDV-Probleme erstellt und anwendet.

264 Seiten, 26 Bilder
Softcover, Fadenheftung
ISBN 3-8023-**0180**-3

38,— DM

Werner Weinberg

Installieren und Patchen von WordStar

3.0 unter CP/M und 3.4X unter MS-DOS

Obwohl WordStar das weitestverbreitete Textprogramm ist, fällt es vielen Anwendern doch schwer, das Programm an ihre individuellen Bedürfnisse optimal anzupassen. Hier setzt das Buch ein. Im ersten Teil wird die normale Installation besprochen, d.h. die Anpassung an den jeweiligen Monitor bzw. Drucker. Der zweite Teil beschreibt das eigentliche Patchen, d.h. das gezielte Ändern von Bytes im WordStar-Userbereich.

336 Seiten, 119 Bilder
Hardcover, Fadenheftung
ISBN 3-8023-**0175**-7

43,— DM

Verlangen Sie unser neues Verzeichnis „CHIP WISSEN Computerbücher 1987“!

VOGEL Buchverlag Würzburg
Postfach 67 40 · 8700 Würzburg 1

ren Programms mitzuberücksichtigen. Damit werden nicht nur die Programme gegen außen offener, sondern sie sind auch leichter zu warten; jeder Programmteil kann für sich in einer isolierten Umgebung getestet werden und funktioniert dann garantiert auch in der Gesamtumgebung.

Nach der Wahl von Modula-2 für die Entwicklung von Farsight und einigen Berechnungen über Aufwand und Ertrag gerade unter dem Gesichtspunkt eines Einsatzes von Modula-2 wurden sich die Partner einig, Farsight zu einem günstigen Preis herstellen und verkaufen zu können. Aus der Rückschau, vor allem in einer Zeit der niedrigen Softwarepreise, ist es bemerkenswert, daß diese Philosophie von Anfang an verfolgt wurde, besonders wenn man sieht, daß die niedrigen Preise bei anderen Produkten normalerweise nicht zustande gekommen sind, weil die Entwicklungskosten besonders niedrig waren, sondern weil meistens ältere, buchhalterisch schon abgeschriebene Versionen ausgegraben und wieder flott gemacht wurden.

Die Arbeit begann sogleich nach der Festlegung des äußeren Erscheinungsbilds, der Funktionen und der Definitionsmodule. Dazu wurde ein handelsüblicher Modula-2 Compiler eingesetzt, der für den IBM PC vom Vierpaß-Ur-Modula-2 Compiler der ETH Zürich abgeleitet worden war und Modula-2 gut unterstützte. Die Arbeit ging am Anfang recht flott voran, doch bereits nach nicht viel mehr als einem Monat stellte sich heraus, daß die mit zunehmendem Umfang des Programmcodes immer längeren Compilerzeiten unerwartet viel Zeit wegfraßen. Dabei waren schon die ersten Resultate bemerkenswert gut, die gestellten Anforderungen wurden von Modula-2 erfüllt. Gefordert wurde jedoch eine effizientere Entwicklungsumgebung.

An einer amerikanischen Universität wurde zu diesem Zeitpunkt ein Forschungsprojekt beendet, welches einen inkrementellen Compiler mit einem integrierten syntaxgesteuerten Editor enthielt, etwas, das es damals nur auf Großrechnern gab. Für die Entwickler von Farsight war es von Vorteil, daß dieses Projekt auf Modula-2 basierte. (Man muß sich vor Augen halten, daß Amerika zu diesem Zeitpunkt noch kaum etwas von Modula-2 wußte. Erst im August 1984 brachte Byte, das führende amerikanische Mikrocomputermagazin, eine Ausgabe mit dem Thema Modula-2 als Schwerpunkt, nachdem Jerry Pournelle, der bekannte Kolumnist, schon einige Monate vorher seine ersten literarischen Freudensprünge vollbracht hatte, nachdem er Prof. Wirth, den Vater von Modula-2, kennengelernt hatte.) Man zögerte nicht lange und kaufte kurzerhand das ganze Projekt mitsamt den Entwicklern auf.

Innerhalb weniger Monate wurde aus dem Hochschulprojekt ein handfestes, komfortables Modula-2 Software-Entwicklungssystem, M2SDS, Modula-2 Software Development System, mit welchem nun zügig an Farsight weitergearbeitet werden konnte.

Es ist vielleicht nicht uninteressant, zu sehen, daß nicht nur M2SDS einen Einfluß auf die Entwicklung von Farsight hatte, sondern auch umgekehrt Farsight auf M2SDS. Nachdem die Arbeit an Farsight mit neuem Schwung fortgesetzt werden konnte, kam die Idee auf, auch den Modula-2-Compiler mitsamt seinem syntaxgesteuerten Editor und dem Linker in den gleichen Window-Manager wie die Farsight-Komponenten einzubetten. Natürlich war auch M2SDS in Modula-2 geschrieben, und so dauerte es keine zwei Monate, bis M2SDS im neuen Kleid dastand. Hinzu kamen noch ein Bibliotheksmanager, damit die Modula-2-Module wesentlich komprimierter als bisher gespeichert wurden.

Es ist erstaunlich, nach dem Einbau einer Hilfefunktion stand die erste Farsight-Komponente zum Verkauf bereit – nicht etwa Textverarbeitung oder Tabellenkalkulation, wie von Anfang an geplant, sondern ein Modula-2 Software-Entwicklungssystem.

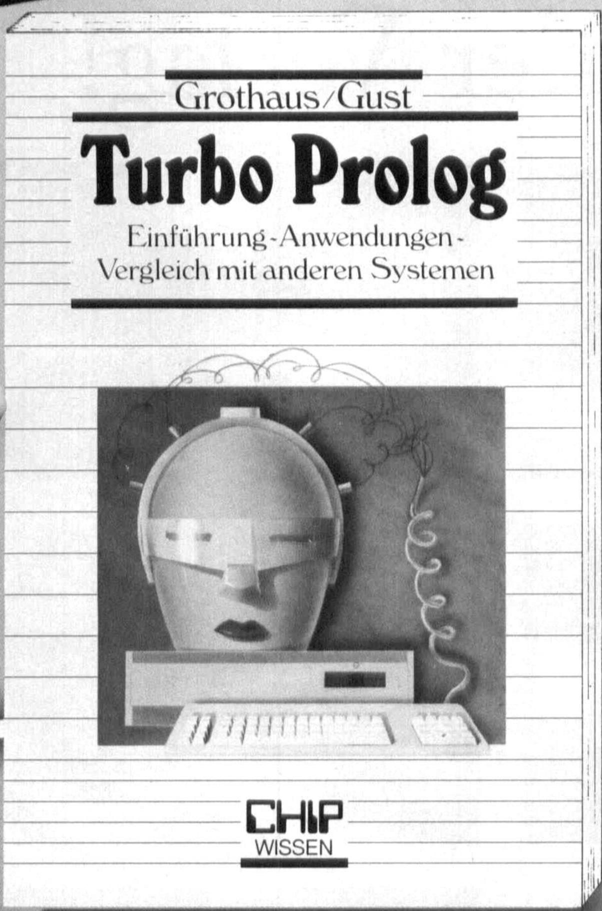
Bei der Präsentation von Farsight im Frühjahr 1986 standen neben der englischen Version von Farsight eine deutsche und eine französische Übersetzung zur Verfügung, die, dank der modularen Aufteilung nicht nur der Komponenten sondern auch der Programme in sich selber, gleichzeitig zu der englischen entwickelt und getestet werden konnten. Modula-2 hatte sich durch und durch bewährt.

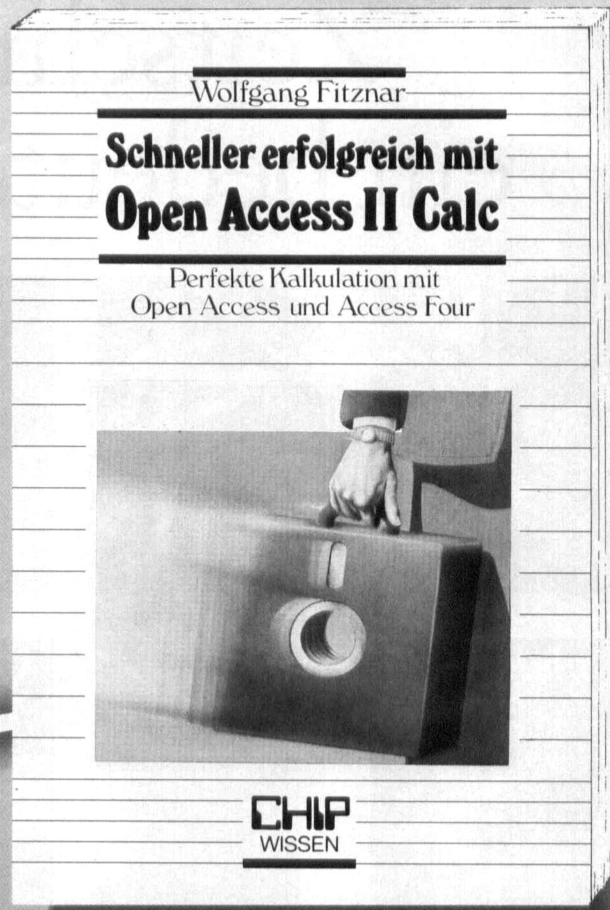
Impressum

CHIP PLUS erscheint monatlich als Beilage zu CHIP. Anschrift der Redaktion: Schillerstr. 23a, 8000 München 2, Tel. 0 89-5 14 93-0, Verlag: Vogel Verlag KG, Max-Planck-Str. 7/9, 8700 Würzburg, Tel. 09 31-41 02-1. Redaktionsdirektor: Richard Kerler. Redaktion: Martin Stübs (verantw.), Gerhard Bader. Grafische Gestaltung: Hans Kuh. Chef vom Dienst: Marianne Weißbach. Verlagsdirektor: Dr. Andreas Kaiser. Anzeigen: Friedrich Mangold. Satz, Litho, Druck, Verarbeitung, Versand: Vogel-Druck Würzburg. Der Verlag haftet nicht für unverlangt eingesandte Manuskripte und Fotos.

CHIP WISSEN

Neuerscheinung





Manfred Grothaus
Helmar Gust

Turbo Prolog

Einführungen * Anwendungen * Vergleich mit anderen Systemen

Das Buch stellt Turbo Prolog vor, beschreibt die Programmentwicklungs-Umgebung und vermittelt grundlegende Programmiertechniken. Die Vor- und Nachteile von Turbo Prolog im Vergleich zu anderen Prolog-Systemen werden aufgezeigt und diskutiert. Alle Teile des Buches enthalten ausführlich dokumentierte Beispielprogramme. Es ist so aufgebaut, daß es als Begleittext zur direkten Arbeit mit dem Rechner verwendet werden kann.

ca. 208 Seiten, zahlreiche Bilder
Hardcover, Fadenheftung
ISBN 3-8023-**0193**-5

38,— DM

Wolfgang Fitznar

Schneller erfolgreich mit Open Access II Calc

Perfekte Kalkulation mit Open Access und Access Four

Dieses Buch unterstützt und begleitet den Anwender bei der Arbeit mit dem Kalkulationsmodul „Calc“ des neuen Open Acess II und dem integrierten Programm „Access Four“. Es stellt sicher, daß der Anfänger in kurzer Zeit seine wichtigsten Aufgaben problemlos erledigen kann. Dem Fortgeschrittenen dient es als Arbeits- und Nachschlagewerk und erspart ihm somit das Suchen in umfangreichen Handbüchern.

224 Seiten, 60 Bilder
Hardcover, Fadenheftung
ISBN 3-8023-**0173**-0

40,— DM

Verlangen Sie unser neues Verzeichnis „CHIP WISSEN Computerbücher 1987“!

VOGEL Buchverlag Würzburg
Postfach 67 40 · 8700 Würzburg 1

Hinter diesen erfolgreichen Zeitschriften steht ein bedeutender Verlag!*



* Über weitere hier nicht abgebildete Titel informiert unser Gesamtprospekt.

Wenn Sie mehr über Vogel Würzburg und seine erfolgreichen Fachzeitschriften wissen wollen, rufen Sie uns an: Tel. (09 31) 41 02-3 46.
Oder fordern Sie mit diesem Coupon ausführliche Unterlagen an. Besonders interessieren mich die Zeitschriften

____

Name, Vorname

Firma | Funktion

Straße/Postfach

PLZ/Ort



VOGELWürzburg

Tradition, Kontinuität, Innovation.

Vogel-Verlag KG
Würzburg
Max-Planck-Str. 7–9,
Abt. 699
D-8700 Würzburg 1